Recurrent Neural Networks (RNN)

Deep Learning (DSE316/616)

Vinod K Kurmi

Assistant Professor, DSE

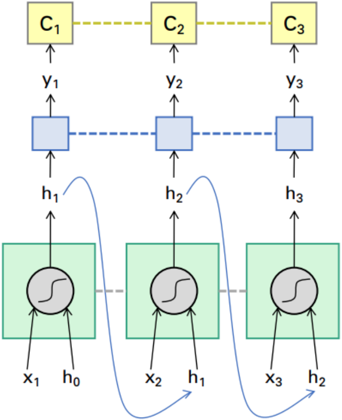
Indian Institute of Science Education and Research Bhopal

Oct 10, 2022

Disclaimer

- Much of the material and slides for this lecture were borrowed from
    - Bernhard Schölkopf's MLSS 2017 lecture,
    - Tommi Jaakkola's 6.867 class,
    - CMP784: Deep Learning Fall 2021 Erkut Erdem Hacettepe University
    - Fei-Fei Li, Andrej Karpathy and Justin Johnson's CS231n class
    - Hongsheng Li's ELEG5491 class
    - Tsz-Chiu Au slides
    - Mitesh Khapra Class notes

Previous class:The Vanilla RNN Forward



$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

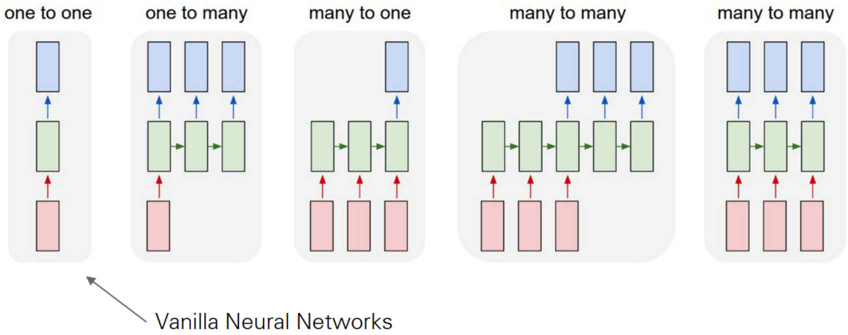
$$y_t = \mathrm{F}(h_t)$$

$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

------ indicates shared weights

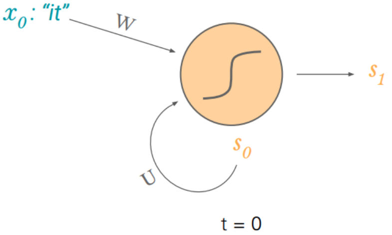
- Note that the weights are shared over time
- Essentially, copies of the RNN cell are made over time (unrolling/unfolding), with different inputs at different time steps

Previous class: Recurrent Networks offer a lot of flexibility



Previous class: Sample RNN

- RNNs remember their previous state:

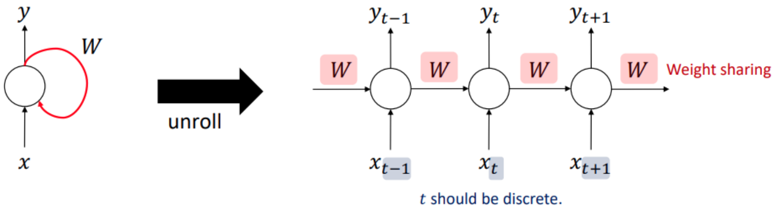


x_0 : vector representing first word
s_0 : cell state at $t = 0$ (some initialization)
s_1 : cell state at $t = 1$

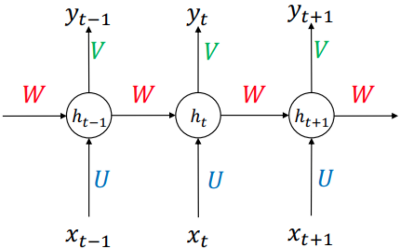
$$s_1 = tanh(Wx_0 + Us_0)$$

$W,\ U$: weight matrices

Recurrent Neural Networks



Recurrent Neural Networks

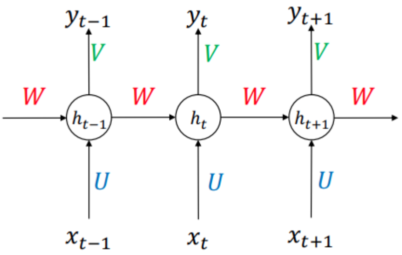


$$h_t = f(Ux_t + Wh_{t-1})$$
$$y_t = f(Vh_t)$$

Backpropagation Through Time (BPTT)

The error using cross-entropy loss, is defined as follows:

$$E(y, \hat{y}) = \sum_t E_t(y, \hat{y}) = \sum_t y_t log(\hat{y})$$



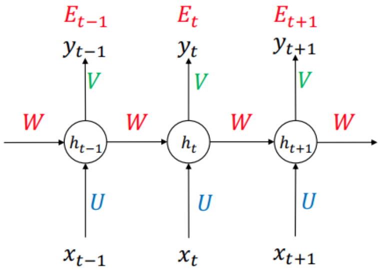
$$h_t = f(Ux_t + Wh_{t-1})$$
$$y_t = f(Vh_t)$$

How can we compute the derivative of the error with respect to W?

Backpropagation Through Time (BPTT)

The gradient for each training instance can be simply computed as the sum of the error at each timestep:

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

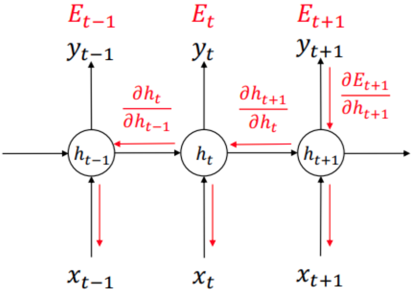


$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial V} = \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial V}$$
$$= (\hat{y}_t - y_t) \otimes h_t$$

The partial derivative of error with respect to V, $\frac{\partial E}{\partial V}$ only depends on the values at each timestep t.

Backpropagation Through Time (BPTT)

However, to compute the gradient for W and U, we need to sum up the contribution of each time step to the gradient.



$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial W}$$

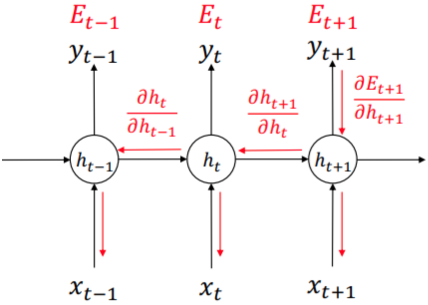
$$h_t = f(Ux_t + Wh_{t-1})$$

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W}$$

Since we use the same W throughout all timesteps, we need to backpropagate through the network all the way to $t = 0$

Vanishing Gradients Problem

The gradient will become smaller as the number of timesteps between the input and the output is larger, as the derivative of *tanH* and sigmoid functions are less than 1

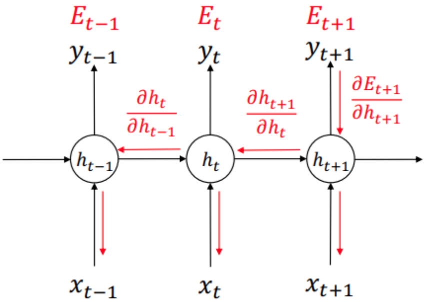


$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j}$$

Exploding Gradient Problem

If the gradients at each step are large, then the gradient will explode, instead of vanishing



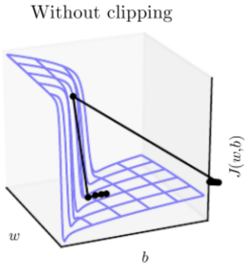
$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

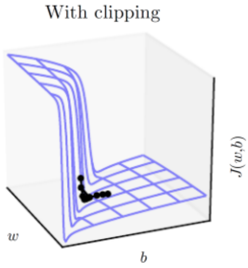
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j}$$

Gradient Clipping

Gradient cliff problem, which causes exploding gradient, can be simply solved by clipping the gradient:

$$\|g\| > v, g \leftarrow \frac{gv}{\|g\|}$$



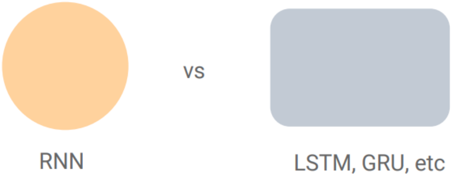


Regularization

- Large recurrent networks often overfit their training data by memorizing the sequences observed. Such models generalize poorly to novel sequences.
- A common approach in Deep Learning is to overparametrize a model, such that it could easily memorize the training data, and then heavily regularize it to facilitate generalization.
- The regularization method of choice is often Dropout.

Gated Cells

- rather each node being just a simple RNN cell, make each node a more complex unit with gates controlling what information is passed through

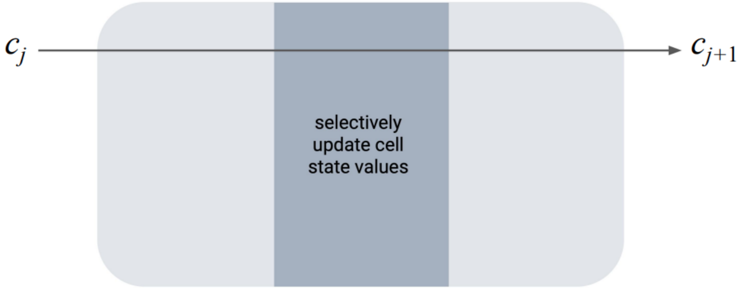


Long short term memory cells are able to keep track of information throughout many timesteps.

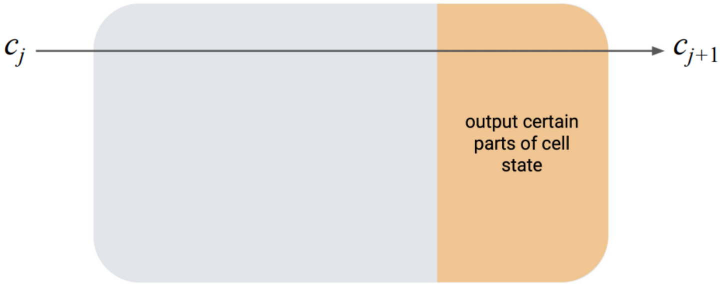
Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM)

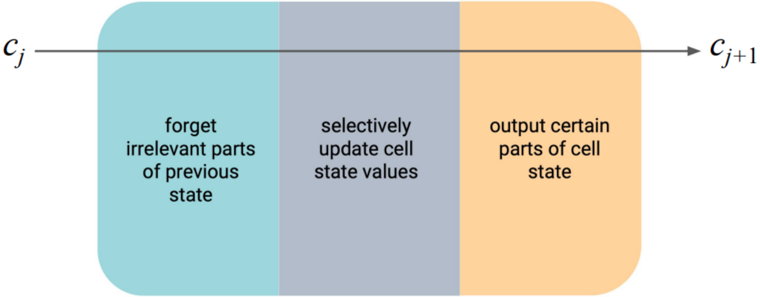
Long Short-Term Memory (LSTM)



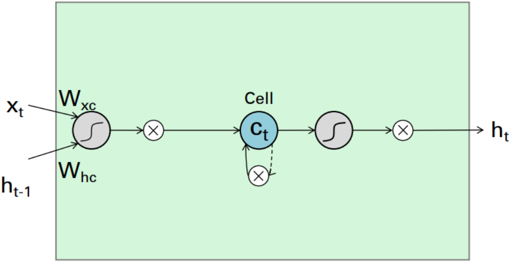
Long Short-Term Memory (LSTM)



Long Short-Term Memory (LSTM)



Long Short-Term Memory (LSTM)

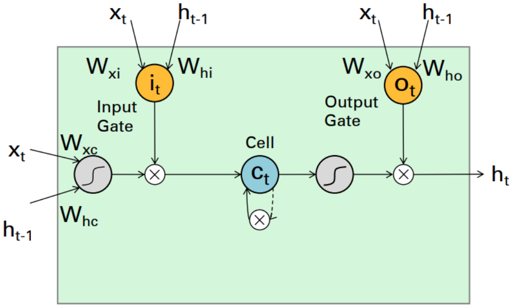


$$c_t = c_{t-1} + \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$h_t = \tanh c_t$$

Long Short-Term Memory (LSTM)

The Original LSTM Cell



$$c_t = c_{t-1} + i_t \otimes \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

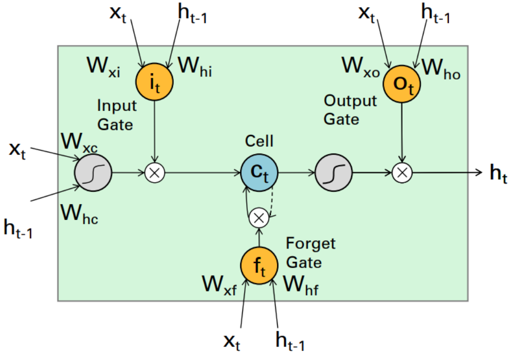
$$h_t = o_t \otimes \tanh c_t$$

$$i_t = \sigma \left(W_i \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_i \right)$$

Similarly for o_t

Long Short-Term Memory (LSTM)

The Popular LSTM Cell



$$i_t = \sigma\left(W_i \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_i\right)$$

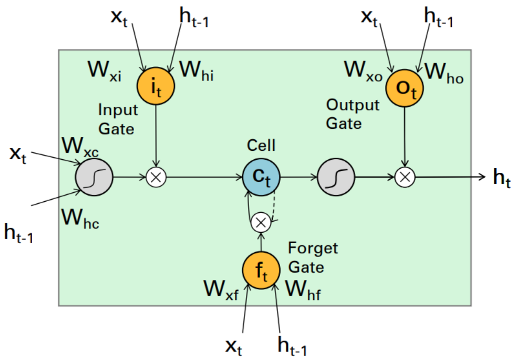
$$c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$f_t = \sigma\left(W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

$$h_t = o_t \otimes \tanh c_t$$

Long Short-Term Memory (LSTM)

The Popular LSTM Cell



$$i_t = \sigma\left(W_i\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix} + b_i\right)$$

$$c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh W\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix}$$

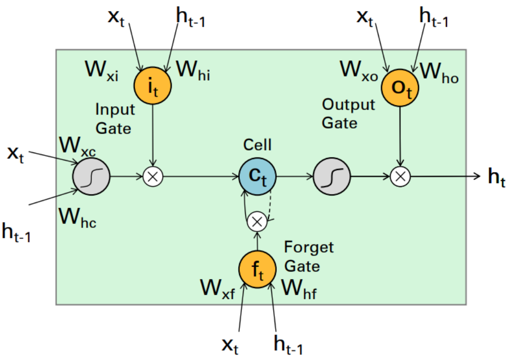
$$\boxed{f_t = \sigma\left(W_f\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix} + b_f\right)}$$

$$h_t = o_t \otimes \tanh c_t$$

forget gate decides what information is going to be thrown away from the cell state

Long Short-Term Memory (LSTM)

The Popular LSTM Cell



$$\boxed{i_t = \sigma\left(W_i \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_i\right)}$$

$$c_t = f_t \otimes c_{t-1} + \boxed{i_t \otimes \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}}$$

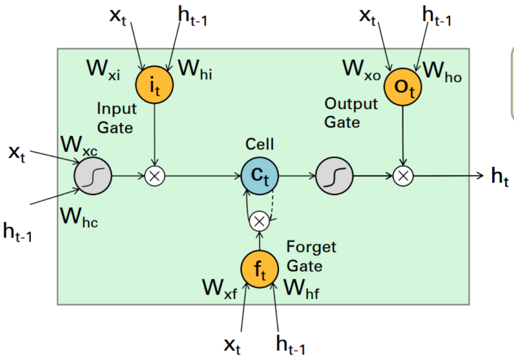
$$f_t = \sigma\left(W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

$$h_t = o_t \otimes \tanh c_t$$

input gate and **a tanh layer** decides what information is going to be stored in the cell state

Long Short-Term Memory (LSTM)

The Popular LSTM Cell



$$i_t = \sigma\left(W_i \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_i\right)$$

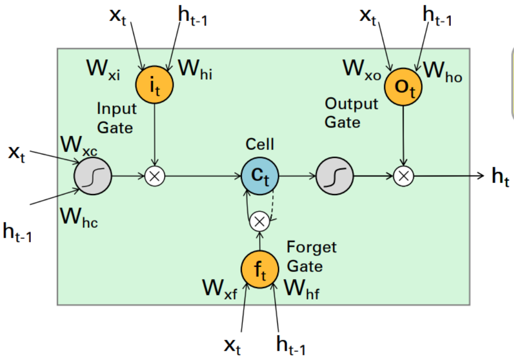
$$\boxed{c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}}$$

$$f_t = \sigma\left(W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

$$h_t = o_t \otimes \tanh c_t$$

Update the old cell state with the new one.

The Popular LSTM Cell

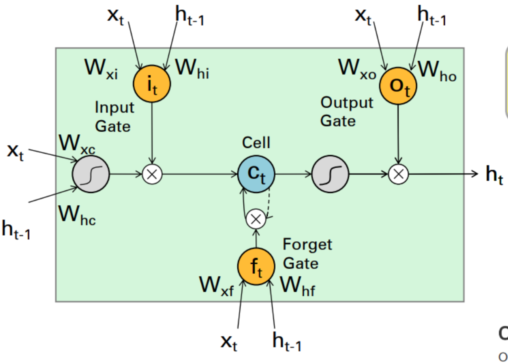


$$i_t = \sigma\left(W_i \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_i\right)$$

$$\boxed{c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}}$$

input gate	forget gate	behavior
0	1	remember the previous value
1	1	add to the previous value
0	0	erase the value
1	0	overwrite the value

The Popular LSTM Cell



$$i_t = \sigma\left(W_i\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix} + b_i\right)$$

$$\boxed{c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh W\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix}}$$

$$f_t = \sigma\left(W_f\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix} + b_f\right)$$

$$h_t = o_t \otimes \tanh c_t$$

$$o_i = \sigma\left(W_o\begin{pmatrix}x_t\\ h_{t-1}\end{pmatrix} + b_o\right)$$

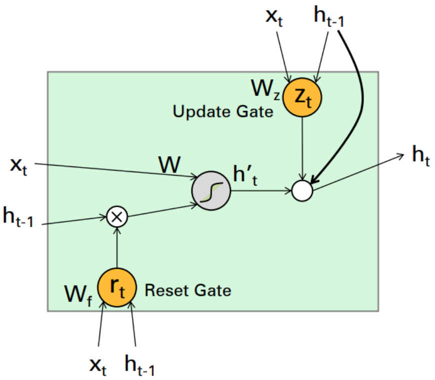
Output gate decides what is going to be outputted. The final output is based on cell state and output of sigmoid gate.

Gated Recurrent Unit

Gated Recurrent Unit

- A very simplified version of the LSTM
  - Merges forget and input gate into a single ‘update’ gate
  - Merges cell and hidden state
- Has fewer parameters than an LSTM and has been shown to outperform LSTM on some tasks

GRU



$$r_t = \sigma\left(W_r \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

$$h'_t = \tanh W \begin{pmatrix} x_t \\ r_t \otimes h_{t-1} \end{pmatrix}$$

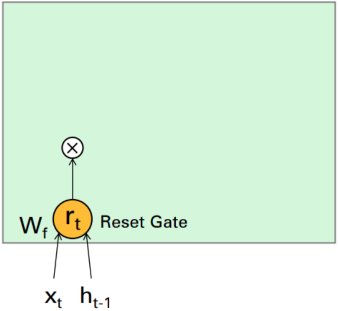
$$z_t = \sigma\left(W_z \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

$$h_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes h'_t$$

Gated Recurrent Unit

GRU

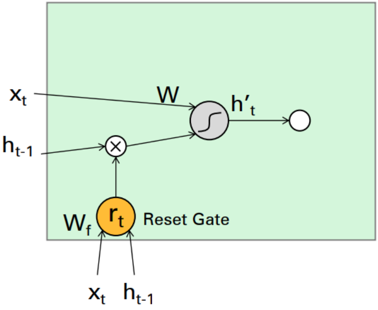
$$r_t = \sigma\left(W_r \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f \right)$$



computes a **reset gate** based on current input and hidden state

Gated Recurrent Unit

GRU



$$r_t = \sigma\left(W_r \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

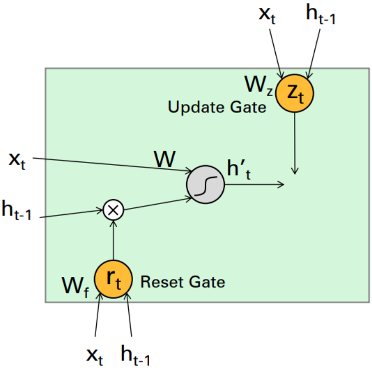
$$h'_t = \tanh W \begin{pmatrix} x_t \\ r_t \otimes h_{t-1} \end{pmatrix}$$

computes the **hidden state** based on current input and hidden state

if reset gate unit is ~0, then this ignores previous memory and only stores the new input information

Gated Recurrent Unit

GRU



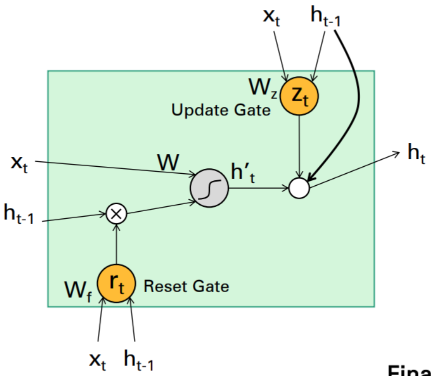
$$r_t = \sigma\left(W_r \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f \right)$$

$$h'_t = \tanh W \begin{pmatrix} x_t \\ r_t \otimes h_{t-1} \end{pmatrix}$$

$$z_t = \sigma\left(W_z \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f \right)$$

computes an **update gate** again based on current input and hidden state

GRU



$$r_t = \sigma\left(W_r\begin{pmatrix} x_t \\ h_{t-1}\end{pmatrix} + b_f\right)$$

$$h'_t = \tanh W\begin{pmatrix} x_t \\ r_t \otimes h_{t-1}\end{pmatrix}$$

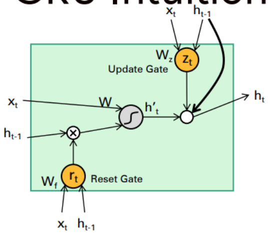
$$z_t = \sigma\left(W_z\begin{pmatrix} x_t \\ h_{t-1}\end{pmatrix} + b_f\right)$$

$$h_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes h'_t$$

Final memory at timestep t combines both current and previous timesteps

Gated Recurrent Unit

GRU Intuition



$$r_t = \sigma\left(W_r\begin{pmatrix} x_t \\ h_{t-1}\end{pmatrix} + b_f\right)$$

$$h'_t = \tanh W\begin{pmatrix} x_t \\ r_t \otimes h_{t-1}\end{pmatrix}$$

$$z_t = \sigma\left(W_z\begin{pmatrix} x_t \\ h_{t-1}\end{pmatrix} + b_f\right)$$

$$h_t = (1 - z_t) \otimes h_{t-1} + z_t \otimes h'_t$$

- If reset is close to 0, ignore previous hidden state
  - ➢ Allows model to drop information that is irrelevant in the future
- Update gate z controls how much of past state should matter now.
  - – If z close to 1, then we can copy information in that unit through many time steps! **Less vanishing gradient!**
- Units with short-term dependencies often have reset gates very active

Slide credit: Richard Socher

LSTMs and GRUs

GOOD

- Careful initialization and optimization of vanilla RNNs can enable them to learn long(ish) dependencies, but gated additive cells, like the LSTM and GRU, often just work

BAD

- LSTMs and GRUs have considerably more parameters and computation per memory cell than a vanilla RNN, as such they have less memory capacity per parameter

Auto-Encoder, VAEs, GANs etc.

Next Class